

wish to share our experience, and report the results of our practical experiments. We concentrate on agent setup, lifecycle, and system interfaces.

Section 2 gives an overview over the way mobile agents are set up in our own mobile agent server SeMoA, and how these agents access server facilities such as mobility.

2. Agent Setup and Lifecycles

SeMoA takes a rather paranoid stance when it comes to setting up agents. A mobile agent is transported by means of a *Java Archive* (JAR), which contains the serialized state of the agent among other data. Before the agent is de-serialized and one of its classes is linked into the server's JVM, the archive's contents have to pass a configurable pipeline of security filters. The filters we provide support agent authentication and integrity checks by means of digital signatures, selected revealing of data with detection of protocol interleaving attacks [13], bytecode filtering, and more.

Once the agent is admitted to the server, a thread group and class loader are created for this agent. A launcher thread is spawned in this thread group, which takes care of de-serializing the agent and later on becomes the first thread of the agent. Although a malicious class of the agent may abuse callbacks in the *Java Serialization Framework* to seize control of the thread in which de-serialization takes place, this does not give the agent access beyond what it would have been granted anyway.

Agents request migration by setting a ticket that points to the desired destination. However, the server transports the agent only after all threads in the agent's thread group have terminated. Again, we verify that serialization callbacks were not used to spawn new threads. The invariant we enforce in this way is that, at the time of agent transport, no class of that agent is on any thread's stack frame any more (unless the agent successfully attacked the systems, e.g. it hijacks the garbage collector thread by means of sneaking a malicious implementation of `finalize()` around our byte code filters).

When the agent instance is de-serialized, its class name is compared to a property that is signed along with the agent's static part by the agent's owner. In conjunction with class signing, this prevents adversaries from substituting the principal agent class with another runnable class that might be included in the agent.

Once, the agent instance is de-serialized, it is passed to a *lifecycle registry* (see figure 1), which passes it subsequently to all registered *lifecycle factories* until a factory signals that it is willing to handle the agent's class. This factory (see also [7] for an introduction to the factory pattern) generates a *lifecycle instance* that can handle the agent's lifecycle, and wraps around the agent instance. The lifecycle in-

stance translates between SeMoA's native lifecycle and the lifecycle of the *acme* system. In particular, it instantiates all necessary components that make the agent instance believe that it is running on its native system.

Subsequent to setting up the lifecycle, the principal agent thread is annotated with four facilities by means of `InheritableThreadLocal` subclasses:

Mobility context: provides methods to set destination tickets, retrieve the agent's name, and access more agent-specific data.

Communication context: provides methods for sending and receiving messages.

Environment: provides dictionary operations on a shared space of objects, with a hierarchical name space for the keys. All operations are subject to access control, and published objects may be wrapped into proxys that implement varying degrees of separation between callers and called objects.

Variables context: provides read access to an agent's properties, taken from the agent's archive.

These annotations are inherited by all threads that are spawned subsequently from the annotated thread. Hence, they are available to all threads of an agent. Access to the annotated facilities is granted based on an agent-specific *tag permission*. This is a permission that is unique for each agent instance in the server, and is assigned only to the classes of that agent. This prevents threads of one agent from accessing the facilities that are assigned to another agent's threads in the case of a direct inter-agent method invocation.

The bottom line of this is that we need not rely on a special agent class in order to make initial system hooks available to the agent, as is common in contemporary mobile agent systems. By default, SeMoA agents must implement only the `Runnable` interface, although any other interface or class could be supported easily as well. Access to primitives such as migration and communication are provided solely by means of thread annotations. These approaches, the one based on an abstract agent class and the one we took, are juxtaposed in figures 2 and 3. From the perspective of interoperability, it is generally advantageous to use interfaces as types rather than abstract agent classes, because this allows an agent class to maintain type-compatibility with multiple systems simultaneously.

For instance, we implemented a lifecycle factory for Jade agents. As a side effect, programmers may write agents based on the abstract Jade agent class, use Jade behaviors, and still access SeMoA's facilities. Such Jade agents are instantly mobile, and benefit transparently from SeMoA's migration and security mechanisms.

3. Case Studies

The first step when integrating support for an *acme* system is, of course, a thorough analysis of that system's architecture and agent lifecycle. Ideally, the *acme* system is available in source code, with appropriate documentation. To some degree, reverse engineering tools are helpful, in particular those that can generate UML [15, 11] diagrams from Java byte code.

The primary goal of this phase is to distinguish agent support from its concrete implementation. Typically, the analysis starts at the system's abstract agent class. All places must be identified, where this class is invoked. Analysis of these places reveals salient details of the system's agent lifecycle. Special attention must be given to thread handling, and subtle assumptions that are relevant for the agent's functioning. Often, such details or not excessively documented, or the documentation abstracts from the particularities of the implementation.

Whenever a method of the abstract agent class is invoked, its parameters and return values must be analyzed for non-trivial types. Ideally, these parameters are of the following types:

- Interface classes; the use of interfaces indicates that the developers of the *acme* system anticipated alternative implementations of the system's functionality.
- Isolated helper classes without references to other *acme* classes; these classes can often be reused without modification.
- Classes that resemble entry points to self-contained subsystems that need no special adaption and can be used as a whole (in other words, modules that can be treated as a black box).
- Java standard classes

Jade's communication package is a positive example of a self-contained subsystem that can be adapted easily. Where parameters do not fall into one of the aforementioned categories, the situation becomes complicated. The analysis must recurse for these classes, and in the end a decision must be taken to the effect whether the integration is feasible and worth the effort.

The next phase deals with the mechanisms used by agents to access facilities such as migration and communication. The fewer and the more concise these mechanisms are, the easier is it to emulate the *acme* system. Systems that clearly define a limited set of interfaces to this purpose are easier to interoperate with than systems that have dependencies scattered all over the implementation.

Adaption of agent communication has the specific problem of addressing peer agents correctly. This is less

troublesome for *acme* agents that are created on a SeMoA server. Many agent systems use naming schemes based on the *Uniform Resource Locator* [2] syntax, for instance something like `wombat@gwork.org:40000/strangeplace`, where "wombat" is a name that can be chosen freely by the agent's creator. However, SeMoA allows no free choice of an agent's name, instead an agent's name is computed implicitly¹ from a digital signature of its static part (see [12] for reference). If the agent is created on the SeMoA server, and its implicit name is computed as `f42a1cc0` then the agent can be given the name `f42a1cc0@gwork.org:40000/strangeplace` in order to match *acme*'s syntax.

If the agent has its origin elsewhere, and is assigned a human readable name such as "wombat", then a suitable mapping mechanism must be used by the lifecycle implementation in order to translate back and forth between these names as required.

Our experience up to the time of writing shows that agent communication is less of a problem when compared to agent migration, though. Migration is often more tightly interwoven in a system's design and implementation. For instance, SeMoA's security policy requires that migration is initiated only after all threads of the migrating agent have terminated, a fact that is hardly taken into consideration by programmers of *acme* agents. However, we do not wish to sacrifice our security policy to interoperability. Termination before migration prevents agents from repeatedly spawning copies, and refusing to terminate afterwards, thus effectively flooding a network of agent servers. It is worth noting that the *full mobility protocol* defined by FIPA enables this type of attack by virtue of its specification, and requires that at least the problem of asynchronous thread termination is solved satisfactorily. Consequently, a lifecycle implementation might have to defer execution of a `go()` statement to the point where it got rid of stale threads that were spawned in the agent's thread group either by the agent itself or as a consequence of a call to, for instance, Java's *Abstract Window Toolkit* (AWT).

In summary, the *acme* system is probably straightforward to adapt if it:

- is available in source code, and well documented (no surprise here);
- confines the dependencies between agents and the system to a clear and well-defined set of interfaces;
- has a modular design, and anticipates alternative implementations for its modules;

¹Implicit names consist of SHA-1 [6] digests, hence are 20 bytes long. For ease of reading, we give only 8 hexadecimal nibbles rather than the whole 40.

- models facilities that are required by mobile agents separately;
- provide features and services in the form of agents rather than specialized classes which must be adapted or treated in special ways;
- pays attention to security.

To some degree, security has similar requirements as interoperability – in both cases there shouldn't be too many drawbridges that lead into and out of your fortress, because you have to put guards in front of each.

4. Jade

We used Jade Version 2.01 beta as the basis of our experiments with Jade. Jade has a focus on agent communication and cooperation rather than mobility. Consequently, the communication support is well developed whereas migration has only marginal support. Our goal was to run Jade agents in SeMoA, without recompilation, and in a way that allows Jade agents to communicate with other Jade agents, where the peer agent can be either at the same server or at a remote server.

The adaption of Jade was surprisingly straightforward, and did not cause major problems. Jade agents are initialized with a so-called `AgentToolkit` implementation that functions, from an agent's point of view, as the principal hook into the agent system. `AgentToolkit` is actually an interface; the `JadeLifecycle` we developed for SeMoA implements this interface, and mediates between SeMoA and the Jade agent. All mappings could be handled in the `JadeLifecycle`. Figure 4 shows an UML diagram of the classes involved. Bold class names denote classes of SeMoA, all other classes were taken from Jade.

Jade supports scheduled behaviors. This allows agents to periodically repeat a specific action, or be invoked at particular times. This requires managing a global timer and dispatcher thread, which is a responsibility of the `AgentToolkit` implementation, and posed no difficulty.

Communication in Jade bases on CORBA [4], and is well separated in a self-contained package. Our `JadeLifecycle` reuses this package. At boot time, SeMoA activates a message stub that is responsible both for dispatching incoming messages as well as relaying outgoing messages. The stub also takes care of translating from internal to external addresses and vice versa. Our tests confirmed that intra- and inter-platform communication between Jade agents works fine. Furthermore, inter-platform also works fine between agents at Jade servers and SeMoA servers with Jade support.

There is hardly any criticism we could raise on Jade's design with regard to our aims. A minor nuisance was

caused by some classes that were declared as `protected` or `package private` without obvious reason, among them `AgentToolkit`. We changed the access modifiers to `public` and recompiled these classes. Apart from this, we had to make no changes.

5. Tracy

The integration of Tracy had been done with the most recently version 0.54 alpha. At time of writing, Tracy agents in a SeMoA are able to run, communicate, and migrate.

In general the architecture of the Tracy adapter classes is very similar to Jade. Figure 5 illustrates the design of our `TracyLifecycle` and related classes as a UML diagram. Again, bold class names denote classes of SeMoA. The diagram shows only a subset of the classes we developed in order to support Tracy in SeMoA. Tracy distinguishes between mobile agents and system agents. The first ones are able to migrate, the latter have special privileges, e.g. to open a graphical user interface. Both inherit from the abstract base class `Agent`, which we access from the lifecycle class in order to control the agent.

The implementation of the inter-agent communication mechanism was straight forward. Tracy agents communicate by means of a blackboard. The blackboard acts as a hierarchical name space where agents can deposit objects. We wrapped a single `Blackboard` instance in a SeMoA service and published it at boot time in the environment. All instances of `TracyLifecycle` access this single blackboard service whenever an agent wants to read or write messages.

Tracy also uses an intra-agent communication mechanism which allows an agent to send messages to itself. This is used to control the agent's state of execution. In order to adapt this behaviour, the lifecycle registers itself as listener at the agent's message queue. In consequence the lifecycle is able to intercept and proceed all messages of the particular agent instance.

More difficult than support communication was to adapt Tracy's migration metaphor. Whenever an agent wants to migrate, it throws a `WantToMigrate` exception. By assumption, this kind of exception may not be caught within the agent. It is caught by a server thread, which processes the agent's request. In order to support this behaviour, the `run` method of `TracyLifecycle` is realized as a loop. In accordance with SeMoA's security policy, an agent has to throw the exception² and terminate all spawned threads.

The adaption of the migration process itself was very simple. SeMoA's architecture supports different transport mechanisms, which are distinguished by protocol part of

²Besides the `WantToMigrate` exception, Tracy also defines the `WantToDie` exception, which indicates that the agent wants to terminate.

the target's URL. Because Tracy agents always use the protocol `tracy` for migration, we simply published a handler instance for this protocol. In fact, this handler actually uses a simple socket connection for transport. This allows the agent to migrate between different SeMoA hosts. We soon will provide a protocol handler for the real Tracy migration protocol in cooperation with the Tracy authors, Peter Braun and Jan Eismann.³

6. Conclusions

In this paper, we presented a bottom-up approach towards interoperability of mobile agent systems, which is based on voluntary interoperability between selected agent systems, rather than a top-down approach driven by standards.

In particular, we presented a number of design approaches that facilitate the incorporation and support of agents of other systems in our own mobile agent server SeMoA. One of the key features of our design is the modeling of agent lifecycles by means of specialized lifecycle implementations that translate between a native lifecycle and the lifecycle of adapted systems. Lifecycle implementations have the task to make agents believe that they are running in a native environment although they actually do not.

In the course of pursuing interoperability between different mobile agent systems we gained considerable insight both in the particularities of Java as well as in the do's and do not's of mobile agent system design.

At the time of writing, our work is far from complete, yet we can already demonstrate a successful integration of Jade and Tracy agents. Furthermore we work at an integration of Aglets [8], a popular mobile agent system. Our future work will address migration protocols of *acme* systems, so that agents are able to migrate back and forth to and from SeMoA and *acme* systems.

References

- [1] F. Bellifemine, A. Poggi, and G. Rimassa. Jade programmers guide, June 2000. Available at URL <http://sharon.cselt.it/projects/jade>.
- [2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Request for Comments 1738, Internet Engineering Task Force, December 1994.
- [3] P. Braun, C. Erfurth, and W. R. Rossak. An introduction to the Tracy mobile agent system. Technical Report No. 2000/24, Friedrich Schiller University of Jena, Computer Science Department, September 2000. Available at URL <ftp://ftp.minet.uni-jena.de/ips/braun/bericht-00-24.pdf>.
- [4] CORBA 2.6 specification. Technical Report formal/20011235, Object Management Group, 2001. Available at URL <http://www.omg.org>.
- [5] FIPA agent management support for mobility specification. FIPA document PC00087A, Foundation for Intelligent Physical Agents, Jun 2000. Available from URL <http://www.fipa.org/specs/00087/>.
- [6] FIPS180-1. Secure Hash Standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, April 1995. supersedes FIPS 180:1993.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns*. Addison Wesley Longman Publishing Co., December 1994. ISBN 0201633612.
- [8] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents With Aglets*. Peachpit Press, September 1998. ISBN 0201325829.
- [9] L. Magnin, V. T. Pham, A. Dury, N. Besson, and A. Thieffaine. Our guest agents are welcome to your agent platforms. Submitted for publication.
- [10] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Omo, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF – The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer Verlag, Berlin Heidelberg, September 1998. The MASIF specification is available at URL <http://www.fokus.gmd.de/research/cc/ecco/masif/doc/97-10-05.pdf>.
- [11] J. Rambaugh, I. Jacobson, and G. Booch. *Unified Modelling Language Reference Manual*. Addison Wesley, 1999. ISBN 020130998X.
- [12] V. Roth. Scalable and secure global name services for mobile agents. 6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages (Cannes, France, June 2000).
- [13] V. Roth and V. Conan. Encrypting Java Archives and its application to mobile agent security. In F. Dignum and C. Sierra, editors, *Agent Mediated Electronic Commerce: A European Perspective*, volume 1991 of *Lecture Notes in Artificial Intelligence*, pages 232–244. Springer Verlag, Berlin, 2001.
- [14] V. Roth and M. Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.
- [15] Unified modeling language (UML), version 1.4. Specification formal/20010967, Object Management Group, 2001. Available from URL <http://www.omg.org/>.

³Currently Tracy experiences redesign within the transport layer.

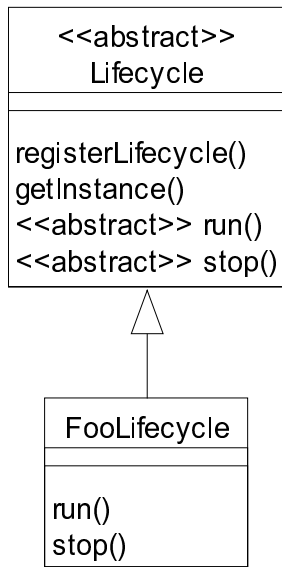


Figure 1. In SeMoA, an agent's lifecycle is modelled as a factory pattern rather than hardcoded into the agent system.

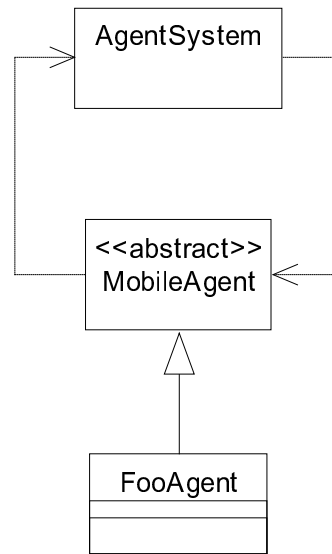


Figure 2. Agents extend a well-known abstract class that provides the basic hooks into the system.

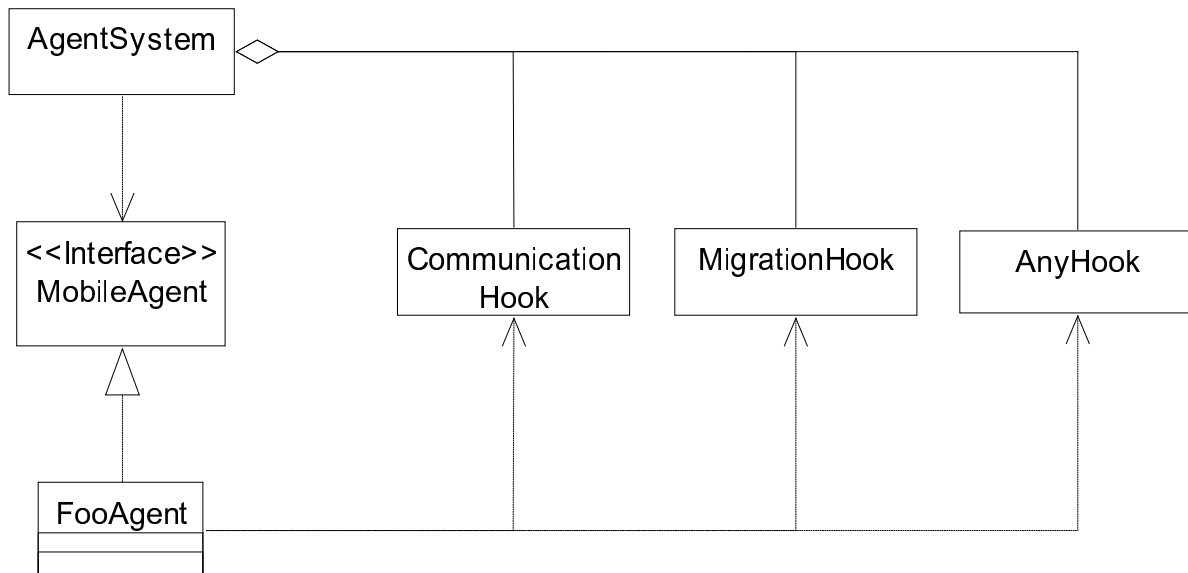


Figure 3. Agents implement an interface as the primary type. Hooks into the hosting system are modelled as separate facilities. In the case of SeMoA, an agent's threads are annotated with facilities such as agent mobility, communication, and access to shared object instances.

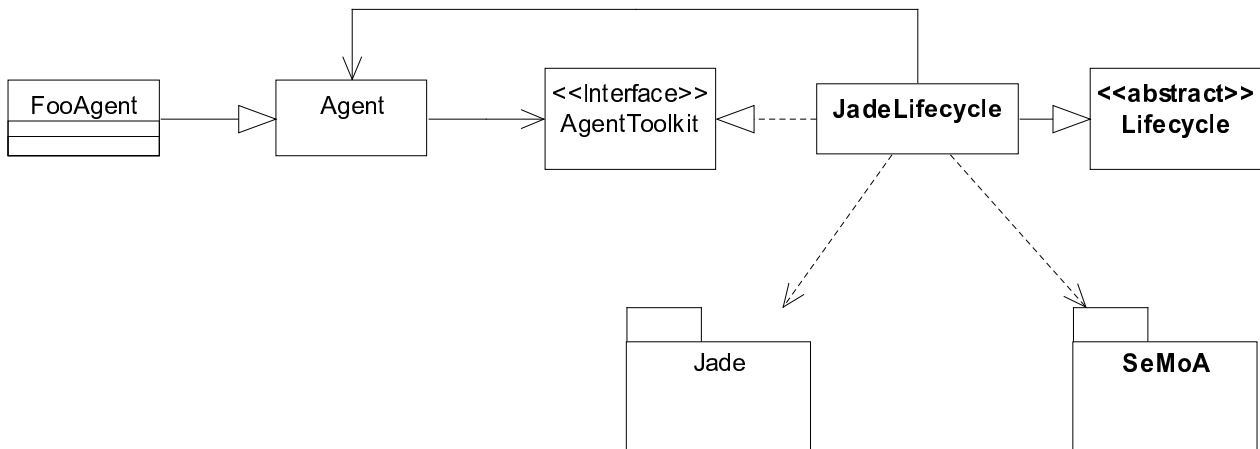


Figure 4. UML class diagram showing an implementation of the lifecycle pattern for interoperability with Jade agents.

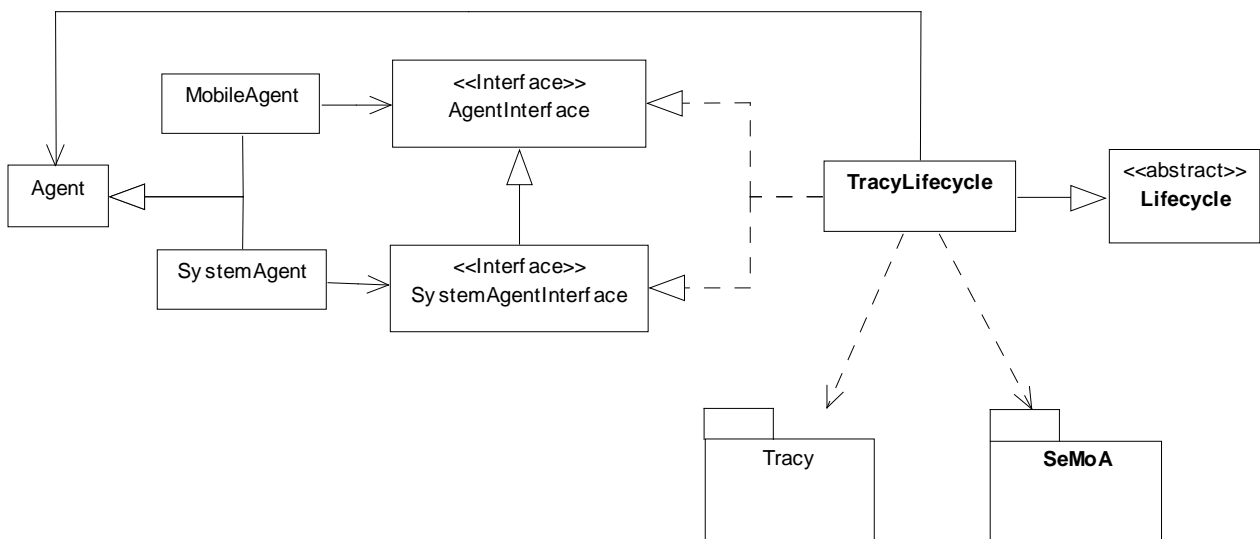


Figure 5. UML class diagram showing an implementation of the lifecycle pattern for interoperability with Tracy agents.